FERMILAB-Conf-87/162

# User Friendly Far Front Ends*

J.R. Zagel and L.J. Chapman
Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

October 1987

*Submitted to the Europhysics Conference on Control Systems for Experimental Physics, Villars-sur-Ollon, Switzerland, September 28-October 2, 1987

# USER FRIENDLY FAR FRONT ENDS

J. R. Zagel and L. J. Chapman
Fermi National Accelerator Laboratory *
Mail Station 307, P.O. Box 500
Batavia, Illinois, 60510, USA

## Abstract

For years controls group hardware designers have designed microprocessor systems to accomplish specific functions such as refrigeration control. These systems often require years of software effort. The backlog of requests for even simple additions or modifications to software in embedded controllers has become overwhelming. We discuss the requirements of "far forward" controllers. Can we build a system that is modular enough to be configurable for specific applications without sacrificing performance? The hardware and software framework must be complete enough to isolate the end user from the peculiarities of bit and byte manipulation, but still allow system specifics to be implemented by the user. It is time to provide embedded controller systems as easy to use as workstation consoles.

## Historical Perspective

Evolution in data collection methodology has provided new challenges and opportunities. Only a few years ago an engineer would design a very specific collection of logic gates and analog components into a module capable of a single simple function. With the advent of microprocessors and analog function modules the far-forward data collection point now provides the services of what was once the entire control system. While the hardware designer now finds it convenient to combine more functionality into the various modules the bulk of the total control task has shifted to the software designer.

Most early control systems were pure analog. Our present day systems are pushing the digital/analog dividing line closer to the transducers and actuators. The digital interface now is very much a part of the real world device. Hence the control system processor must have an intimate knowledge of the device it controls or monitors.

How do we map what a computer knows (instructions, variables), into the controls problem (valves, temperatures, closed loops)? And how much of that mapping should be done by system software? Conventional tools provide such a poor mapping that it is very difficult to write generic software. The mapping is so complex, and so loose, that the number of possibilities is enormous. This, together with the creativity of programmers, result in a bewildering variety of incompatible approaches. But new paradigms (objects and rules) give a much cleaner mapping from the control system to the controlled system, making generic controls software more feasible.

A classical software problem has been: what code should access what data? Originally any code could access any data, which of course meant that code often accessed data incorrectly, or at least unwisely. Common blocks helped somewhat, in a crude way. Structured programming languages demonstrated the inadequacy of the structured approach. Now object-oriented languages, a gift from the field of Artificial Intelligence, are here. Objects solve the code/data organization problem and are especially relevant to controls programming.

Conventional computers are inherently sequential as was early software (piano rolls, looms) until the arrival of the GOTO statement. When the British invented subroutines, software became less sequential. Multitasking made it less sequential still, and now rule-based languages, another gift from AI, are arriving, providing a much closer match to the controls problem than more sequential methods possibly can.

## Definition

The User Friendly Far Front End (UFFFE) is the collection of hardware and software that will be provided to allow some useful control task at, or close to, the controlled or monitored system. Its task is any subset of data acquisition, conversion, closed loop processing, and control output. One of the main goals then is to identify the parts of the system that are hard architecture and that can be provided as generic, standard modules. These modules can be provided by the system thus freeing the applications programmer to concentrate on the specific control task.

The hardware in the UFFFE most certainly includes many intelligent modules that can be programmed for various applications. The software to initialize, operate, and recover from fault conditions is a part of the module design and must be provided as a package that other software can access. How to use the data is an application question whereas the acquisition is still specific to the hardware involved.

## Generic Resources

### Hardware Objects

We have begun providing modules that accomplish ever more complex functions. But we have fallen into the trap of leaving out the software required to access them. This coding then becomes part of the task of every system programmer who typically does not want to be concerned with the peculiarities of bit and byte manipulation.

A new stage of design is required that specifically devotes attention to setting up, collecting or distributing data, and fault reporting on a given hardware module. A hardware module together with its software is what we call a hardware object.

All of the hardware present in today's control systems can be loosely classified as analog to digital converters, digital to analog converters, digital input, digital output, controllers, and communicators. Questions of how many channels, what speed, and what accuracy or resolution give rise to the plethora of installation variations. This will not change in the future. However it is possible to think in terms of providing, for instance, a standard fast A/D, multiplexed A/D, etc., all complete with the software required to access (and even filter) data, making it available to the system.

* Operated by Universities Research Association Inc., under contract with the US Department of Energy.

## Software Objects

A software object is a little package of data and code which accesses its data. Other objects cannot access this data directly; rather they send messages to the object which result in the object accessing its own data in its own way. These messages provide a clean black box interface around each object. No object knows more about its neighbors than it needs to know. Code is more strictly partitioned and therefore easier to write and maintain, allowing more generic code.

Objects can be used for many purposes in a control system. We have already described hardware objects. Standard control algorithms like digital filters and closed loops can be represented as software objects. All these objects can send messages representing data flow to one another: the temperature sensor object gets the current temperature, sends it to the digital filter object which averages it with recent temperatures and sends the average on to the PID loop object, which calculates a new valve position and sends it on to the relevant valve object (see Figure 1). Thus objects can easily and naturally represent data flow networks, which can easily and naturally represent many control problems.

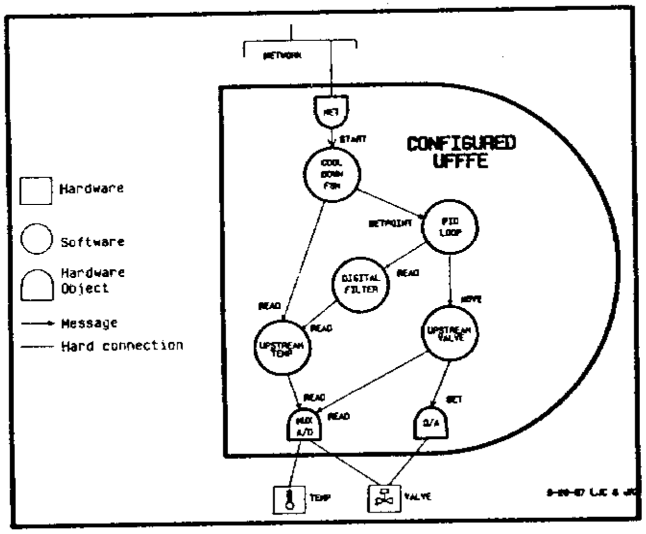

Figure 1. Cryogenic UFFFE

The PID loop object in this example can be coded generically because of the clean interface provided by messages. The loop might obtain its input value by sending "read" messages to any object that supports "read" messages, and similarly send "set" messages to any object that is setable. This would be difficult to code in a conventional language. Object-oriented programming makes it simple to code generically.

Finite state machines provide another example. Many control sequence problems are naturally expressed as state machines. Objects can easily represent such machines, the state being remembered by the machine object. Again this can be done in a very generic way. Also, these two fundamental but high-level paradigms, data flow and state machines, can be combined in various ways; typically a state machine watches over and controls a data flow network, changing loop parameters and opening and closing channels as the state changes. An informal survey within the Fermilab Controls Group showed that several control systems written in radically different styles by various programmers boil down to a state machine supervising a data flow network. If generic support for these paradigms had been available, these projects could have been done in much less time. Similar claims have been made often in the past, but object-oriented programming should make the claims fulfillable.

The messages objects send one another can be sent across communications networks. The network structure, indeed the existence of the network itself, is unknown to the objects, which see the universe as a large collection of objects able to send messages to one another. It should therefore be easy for a closed loop object to run entirely within one processor, or to get its reading from (and send its setting to) objects in other processors. Again this allows very generic code.

## Rules

In rule-based programming, rules replace tasks and subroutines. Each rule is like a tiny task, waiting for conditions to be just right for it to fire, executing some actions. Rules communicate with each other via facts. When a rule's patterns match existing facts, the rule fires, most typically asserting new facts which may eventually cause further rule firings.

Facts provide a clean, flexible, powerful interface among rules. A rule whose pattern matches a fact neither knows nor cares who generated the fact. Conversely, a particular fact can be of interest to any number of rules. This "fact interface" makes it possible for rules to be very independent. Extremely generic facts and rules are possible.

The rule based paradigm is especially appropriate for real time controls systems where, as conditions change over time, various actions should be taken. Such actions can be coded straightforwardly as rules. This structure seems much closer to the controls problem than any more sequential approach.

## Application-Specific Resources

Generic resources are combined to create specific applications. At this level the application designer defines the types of loop feedback desired, the input variables, output variables, and possibly the frequency of updates for the loop. The peculiarities of a particular A/D board, such as multiple readings on a given multiplexer channel, are of no concern to the application. The application has every right to assume that a datum is as accurate as the function module can provide or that the module has reported an error. The designer can work in a higher level, concentrating on the control problem.

Hardware is available to accomplish many input and output functions. There always seems to be some requirement which rules out the use of any existing module. However there is now a significant selection of off-the-shelf equipment that is well defined in both hardware and software. New apparatus can be designed to fit the present interface scheme with only the additional features added.

When a new type of object is required, say a valve with a new parameter, it need not be coded from scratch. Since object types are hierarchical, it is very easy to specify that the new kind of valve is just like the standard valve with one exception. What all valves have in common is coded in one place; what is unique to the new valve is coded in one other place. This allows powerful, generic system code representing what is common to all control systems. Specific applications can begin with the generic code and add ONLY the code

```
Facts:
======
(value us-temp 9)                    The up-stream temperature is currently 9.
(value ds-temp 3)                    The down-stream temperature is currently 3.
(maximum-value us-temp 10)           Up-stream temperature should be less than 10.
(cares console-12 ds-temp)           Console 12 has alarms enabled for the
                                      down-stream temperature.

Generic alarm rules:
====================
(defrule going-high
        (maximum-value ?var ?max)          If a variable has a maximum,
        (value ?var ?val&>?max)            and that variable's current value
                                           is greater than its maximum,
    =>                                     then
        (assert (too-high ?var)))          that variable is too high.

(defrule going-out-of-bounds
        (or                                If a variable
            (too-high ?var)                is too high
            (too-low ?var))                or too low,
    =>                                     then
        (assert (out-of-bounds ?var)))     that variable is out of bounds.

(defrule notify-interested-consoles
        (out-of-bounds ?var)               If a variable goes out of bounds,
        (cares ?c ?var)                    and some console cares about it,
    =>                                     then
        (send-message ?c ?var              tell that console.
         going-out-of-bounds))

A very specific, easy-to-add rule:
==================================
(defrule us-temp-too-high
        (too-high us-temp)                 If the up-stream temperature is high
        (off wet-engine)                   and the wet engine is off,
    =>                                     then
        (open us-valve))                   open the upstream valve.
```

Figure 2. ART[1] Coded Example

necessary to describe what is unique to that application.

One of the chief advantages of rule-based systems is the independence of the rules: ideally each rule expresses a single concept. Rules, and therefore concepts, can be added or removed independently. Some rules can therefore express very generic concepts and be part of the system: concepts like periodicity or alarms. Other rules can express application specific concepts like "whenever the up-stream temperature is too high and the wet engine is off, open the up-stream valve" (see Figure 2). Such rules can be added easily because rules are independent and communicate via facts. Facts such as "temperature 7 too high" can trigger generic rules such as "when anything is too high, notify anyone who cares" as well as application specific rules like the one sketched above.

## Structure

The UFFFE could be assembled using the components described above. The task before us is to provide a system with a processor, communications capability, a varied selection of input/output capabilities, and the software modules to tie these components together.

An example of this process can be found in the personal computer. Here a collection of processor, communication, mass storage, keyboard, mouse, and video display is assembled with at least an operating system to get the system running. Starting with this base a host of specialized applications code is written to accomplish a number of various tasks.

An UFFFE is a collection of hardware and software modules to accomplish a specific task. As such it may require the services of a network to move data. When we provide the hardware that cares about the intricacies of signal levels and bit rates, we also provide the networking software to initialize the hardware and move data. This is true regardless of the actual media. In time our generic system would accommodate any standard networking scheme. Since the interface to UFFFE is well specified any new networking scheme can easily be added as the need arises.

## Conclusion

The advent of new software paradigms and modern hardware finally make it possible to design truly modular systems with truly generic components. Hardware modules and their associated software are supplied as complete objects. Clean interfaces make it easy to add both generic and application-specific objects to an UFFFE, which itself presents a clean interface to the outside world.

## References

[1] ART - Automated Reasoning Tool. Trademark of Inference Corporation. 5300 W. Century Blvd., Los Angeles, California, 90045, USA.